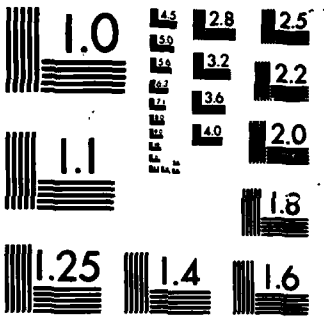
AD-A163 817 REGULAR EXPRESSION ANALYSIS OF PROCEDURES AND EXCEPTIONS(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) J M FOSTER JUN 85 RSRE-85008 1/1

UNCLASSIFIED DRIC-BR-97966 F/G 12/1 NL



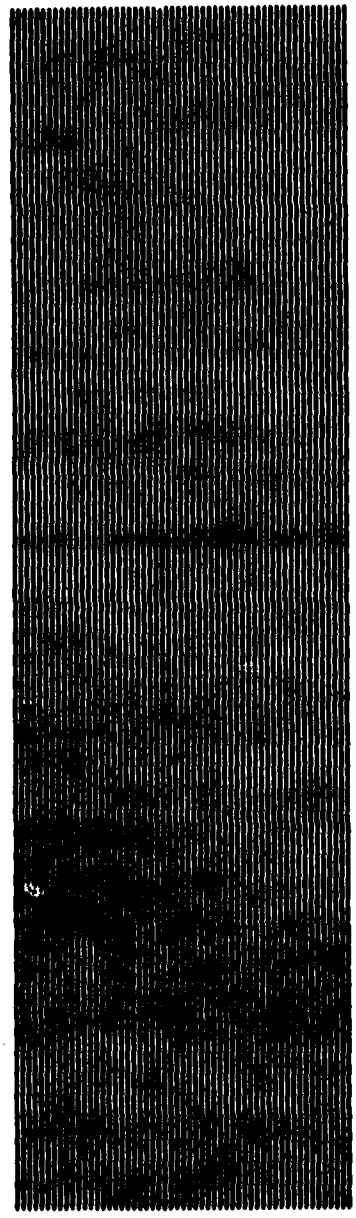
Report No. 85008

Report No. 85008

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

REGULAR EXPRESSION ANALYSIS OF PROCEDURES AND EXCEPTIONS

Author: J M Foster



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

June 1985

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 85008

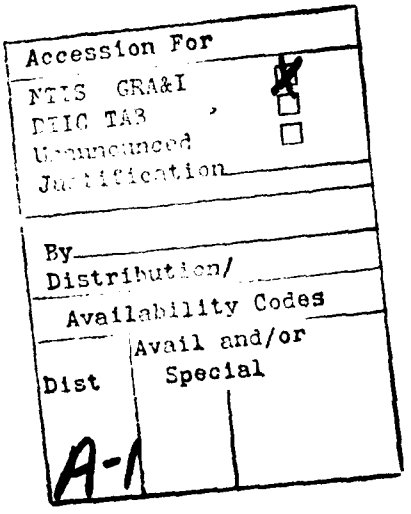
TITLE: REGULAR EXPRESSION ANALYSIS OF PROCEDURES AND EXCEPTIONS

AUTHOR: J M FOSTER

DATE: JULY 1985

SUMMARY

This paper describes an algorithm used to find properties of programs by using algebraic methods. It is capable of handling subroutines and exception exits. A program scheme has been produced to implement it which is particularly aimed at finding properties of microcode programs. This scheme can be systematically instantiated to deal with particular properties. Great care has been taken with the efficiency of the algorithm; it works in practice, even for very complex microcode, in time proportional to the number of microcode instructions. Both the mathematical basis of the algorithm and the algorithm itself are explained.



1 Introduction

Many properties of programs can be found by considering all the possible paths through a program - usually infinitely many - and combining properties calculated for each of these paths. It is possible to develop a theory of the algebra of such paths and the way in which they are combined, and to determine properties of the program by using homomorphisms defined on this path algebra. Such techniques have been discussed particularly in relation to compilation, in order to show the applicability of certain optimisations, but they can also be used to show that various kinds of error are absent from the program (Aho 1977, Allen and Cocke 1976, Backhouse and Carré 1975, Bramson and Goodenough 1982, Bramson 1982, Fosdick and Osterweil 1976, Kam and Ullman 1977, Tarjan 1981, Wegbreit 1975).

It is especially useful to apply these methods to the discovery of errors in microcode (Foster 1984). Because of the great need for speed and shortness of code, microcode is often written in a very dense style, with many jumps and a very complex structure. Errors are easy to make and difficult to find. It is altered less frequently than most programs and every program run on the machine depends on the correctness of the microcode, so the economic costs of checking can be borne. The microcode machine is well-defined and fairly simple so there is less doubt about the semantics of the individual microcode instructions than there is about some constructions in ordinary programming languages. Microcode is unlikely to contain recursive procedures. Foster 1984 gives examples of homomorphisms applied to finding a number of apparently quite different and useful properties of microcode. These include ensuring that an expression stack cannot overflow, finding the maximum time between polling for interrupts, checking that timing constraints on the use of store are met and making sure that the interrupt routine does not disturb values in machine registers. In the rest of the discussion only microcode will be considered, though most of what is said could be applied more generally.

The algebraic structure of the paths through a microcode program is defined so that it depends only on the control structure of the

microcode, not on the changes which the instructions make to registers, store and peripherals, nor on the values in the registers. For the purposes of this paper we shall assume that there are no computed jumps, though a further paper will discuss this problem. Hence, if we write a program to find errors in microcode by the algebraic method, the main body of the program will depend on the control structure. Only that part which defines the individual homomorphism (calculating a particular property) depends on the operations on registers. It is therefore possible to write a program scheme to process the paths, which will be applicable to microcode controlled by a particular sort of microcontroller, and which will be independent of the non-control parts of the microinstructions. This scheme can then be specialised to calculate a property by defining just that part which specifies the homomorphism.

In order to carry out the computations at a reasonable speed it is necessary to avoid re-calculating the same value many times, which would introduce the danger of an exponential explosion. Indeed if there are n conditional statements in sequence, then there are 2^n possible ways through the sequence and the algorithm must not, and need not, repeat the calculations in that fashion. The microcode which defines the Flex architecture on the Perq computer (Currie, Edwards and Foster 1981) consists of about 5000 instructions. Of these one third are labelled and more than half are conditional jumps. With so many possible paths the calculation must be carried out without superfluous evaluations. It is precisely because we are using homomorphisms that the calculations need not be repeated. An essential part of the definition of homomorphism is that the image of a part of the structure is the same, no matter the context in which it lies. In addition, for the kind of algebra we shall be using (regular algebra), there is a distributive law concerning the composition of paths. This implies that, for a homomorphism, the required property of a path from a label to some other point can be calculated and remembered, and the value so produced can be re-used when another jump to the same label is processed.

A similar technique is applicable to sub-routines, which in microcode do not normally have parameters, nor are they recursive. Once again the

homomorphic image of the paths involved in the sub-routine can be calculated, remembered and re-used at each call. Complications set in when we have to consider "exception" exits from the sub-routine. Commonly it is possible when in a sub-routine to decide to abandon it, remove the link from the link stack and jump to some specified place. This typically occurs in the microprogram when checking the conditions which must be met before obeying a macroinstruction. If the conditions are not met the microcode must deal with the failure. This error may be detected during the processing of a sub-routine and this gives rise to the need for an exception exit. The Flex architecture, being based on capabilities and therefore checking against many possible errors, has a large number of such tests. Indeed about one fifth of the instructions contain conditional jumps for error detection, though not all of these are inside sub-routines.

The purpose of this paper is to show how homomorphisms may be efficiently calculated when sub-routines and exceptions are present. A program scheme has been written which operates in this way on microcode based on the AMD 2910 chip and it has been used to calculate many properties of the Flex microcode for Perq to assist its validation. A simplified version of this program scheme, which treats all the points concerned with sub-routines and exceptions, is used for the discussion.

Section 2 introduces the basis of the algebraic method. Section 3 describes the program scheme.

2 The path algebra

For the algebra of paths we use what is known as a regular algebra. This can be defined abstractly, by giving the operations of the algebra and the laws they must obey. We can also give concrete realisations of regular algebras by specifying the objects manipulated and defining the operations on them and then showing that the operations do indeed satisfy the necessary laws. One such realisation is that of paths through a program.

By a path is meant the various ways of getting from one place to

another. In a program this is the possible sequences of instructions invoked in moving from one point in the program to another. A path may be composed of alternative routes, but in one path the start and end points of all the routes are the same.

In a regular algebra there are three operators, written "dot" or ".", "plus" or "+" and "star" or "*", and one constant "one" or "1". If we denote the underlying set of the algebra by E then

$$\begin{aligned} \text{dot} &: E \times E \to E \\ \text{plus} &: E \times E \to E \\ \text{star} &: E \to E \end{aligned}$$

Interpreted in terms of paths "dot" means composition in sequence, that is a.b denotes the path formed by a followed by b. The constant 1 is a unit for the dot operation, so 1.a = a.1 = a. Considered as a path it has zero length; it is a short circuit between its start and finish. The operator "plus" means alternative paths; following the path a+b means either following a or following b. The interpretation of "star" is a circular path: following t* means following t zero or more times.

A regular algebra is usually defined with a "zero" constant which acts as a zero for the dot operator and identity for the + operator, but we shall not need it and will leave it out.

There are a number of identities relating the operators.

$(a.b).c = a.(b.c)$	associativity of .
$(a+b)+c = a+(b+c)$	associativity of +
$a+b = b+a$	commutativity of +
$a+a = a$	idempotence of +
$a.(b+c) = a.b + a.c$	left distribution of . over +
$(a+b).c = a.c + b.c$	right distribution of . over +
$1.a = a$	1 is a left unit for .
$a.1 = a$	1 is a right unit for .

For the purposes of this paper we shall say that

$$a^* = 1 + a + a.a + a.a.a + \ldots$$

It is possible to define the star operator in other ways (Salomaa 1966).

All these identities can be considered in the light of the interpretation as paths and will be seen to be hold in that interpretation. They are particularly important for the algorithm of this paper, since they enable us to re-order the work of calculation of a homomorphism and thus avoid re-calculation.

For the path algebra, the underlying set, E, contains a number of elementary paths. These can be the names of the individual steps of a route, or in the program interpretation they can be the effects of the individual instructions. The smallest algebra containing the atoms and all the terms which can be generated from them using the operators, in which the identities hold, and in which no two values are equal unless this is a consequence of the identities, is the free regular algebra defined on the set of atoms.

The path algebra is an interpretation of the free regular algebra. So to a given microcode program corresponds a regular expression in terms of the effects of the individual instructions.

Our concern is to calculate properties of programs, that is properties of the set of possible routes from start to finish. A very simple example would be the minimum number of steps through a program (Carré, 1979). Clearly if we take two paths of given minimum length and put one after the other, that is, compose them with the dot operator, then the minimum distance through the result is the sum of the minimum distances through the parts. Likewise, if we compose two paths with the plus operator then the minimum distance through the result is the minimum of the distances through the parts. The minimum distance through a starred path is zero, as is the length of the path 1. An atom has length one. So if we took the regular expression for a piece of program and evaluated it, using integer addition for "dot", integer minimum for "plus", the function producing 0 for "star", 1 for each atom and 0 for "one", then we would get an integer which would give us

the minimum number of steps through the program. It is easy to see that the given functions satisfy the identities that we gave before. So we could evaluate the minimum distance through a part of the program, say a sub-routine, and whenever we want to consider the minimum length through that sub-routine in the calculation of the length through another part of the program we could use that pre-computed length. This is permissible because the functions obey the distributive and associative laws, which say precisely that this is permitted.

The interpretation of the operators of regular algebras in terms of integer values and the operations given above is another representation of regular algebras, this time not a free one. The mapping described above is a homomorphism, and the method of calculating it is just how homomorphisms from a free algebra are calculated.

All this is to say that a particular property of the set of routes can be calculated by evaluating a homomorphism from the regular algebra to one defined by the specified functions. It is because the operations obey the identities that we can carry out re-arrangements of the calculation which avoid repeating work.

It is of course necessary to prove that the functions do satisfy the necessary identities. This is for the user of the scheme to do himself.

It is well known that regular algebras are equivalent to finite state machines. For our manipulations we use a slight extension of a usual notation for finite state machines, a notation which mixes finite state machines and regular expressions in a simple way. We shall, as usual, say that a finite state machine consists of a set of terminal symbols, a set of non-terminal symbols of which one is distinguished, and a set of rules of particular form defining each of the non-terminal symbols. In the usual form of this kind of definition a rule is the union of terms each of which is either just a terminal symbol or is a pair consisting of a terminal symbol followed by a non-terminal symbol. The set of strings defined are those, consisting only of terminal symbols, which can be obtained by sequences of substitutions of non-terminals by their definitions starting with the distinguished non-terminal. It is well known that a set of strings that can be defined in this way can also be

defined by a regular expression on the same terminal symbols, and vice versa.

The minor change that we make to the form of the rules is to allow each of the terms in the union to be either a regular expression in the terminals or such a regular expression followed by a non-terminal. This clearly allows us to define just the same sets of strings as before, but permits simpler manipulations. For example, it can be arranged that in the definition part of any rule no non-terminal occurs more than once, by collecting together the regular expressions preceding occurences of the same non-terminal and combining them with the + operator. Indeed, all the rules will be kept in this form in what follows.

3 The algorithm

We shall consider a representative set of control instructions. These are in fact a sub-set of the AMD 2910 instructions, but any microcode controller is likely to have similar instructions.

1) Next. Control passes to the next instruction in sequence.

2) Conditional jump. Control passes either to the next instruction or to an instruction of which the address is given and known at the time of assembly.

3) Decode. This is the last instruction for the evaluation of a particular macroinstruction. The microcode for the next macroinstruction is started at an address which depends on the value of the code for the macroinstruction.

4) Call. The address of the next instruction is pushed onto a special link stack and control is passed to the instruction of which the address is given. There may be a limit to the depth of the stack. In the case of the AMD 2910 the limit is 5.

5) Return. An address is popped off the link stack and control is passed

to the instruction at that address. The effect of popping more addresses off the stack than have been pushed is undefined.

6) Jump-pop. An address is popped off the link stack and control is passed to the address given in the instruction. The effect of this is to ignore the stacked address and this is therefore the instruction which implements exception handling.

To start with, consider programs made up of only two kinds of micro-instruction, next and conditional jump. To each instruction of the microcode will correspond one terminal - which will stand for the non-control parts of that instruction - and one non-terminal. For the instruction Next, with non-terminal A and terminal a, the defining rule is

$$A = a.B$$

where B is the non-terminal for the instruction following A. For the conditional jump instruction, (c, C),

$$C = c.D + c.E$$

where D is the non-terminal for the following instruction which is the one used if the test fails, and E is the non-terminal for the instruction to which control is sent if the test succeeds. Suppose that a particular instruction serves as the start to the program, the distinguished non-terminal, P. We now have a system of rules which defines a finite state machine and hence a regular expression in the terminals, that is, the non-control parts of the instructions.

Processes for turning a set of such rules into a regular expression are well known. In the extended notation it is particularly easy. Any rule which is not directly circular, that is which does not both define and use the same non-terminal within that very rule, can be substituted into every place where it is used and then removed. Such substitution merely involves writing the definition in place and expanding it out using the distributive rule for regular algebras so that all the terms once again are of the proper form. We also choose to collect together all the terms ending with the same non-terminal, using the distributive

law the other way round, so that each non-terminal occurs only once on the right of the rule. Thus it is possible to eliminate all the non-terminals which are not directly circular except that for P.

If there are now any circular rules they can be put into the form

$$X = x.X + \alpha$$

where x and α are regular expressions and α is of the form $\Sigma y_i Y_i$, where $Y_i \neq X$ for any i. Replace this rule by

$$X = x^*.\alpha$$

and again use the distributive law to reduce this to standard form. Now repeat the sequence of substitutions and removal of circularity until the only rule left is of the form

$$P = e$$

for some regular expression, e. Each substitution step decreases the number of non-terminals involved. Each removal of circularity does not increase the number of non-terminals or circular rules and produces a rule which is either the definition of P or can be eliminated because it is not directly circular. Hence the process terminates. Each step preserves the strings defined by each of the remaining non-terminals, as follows from the regular algebra identities. Therefore e is a regular expression for the set of strings of terminals defined by P in the rules we started from.

A different order of performing these operations might end with a different regular expression but, by the argument above, each order would define the same set of strings. We are not attempting to find the shortest regular expression, such optimisation would not significantly improve the speed of the algorithm and would itself be very expensive to carry out.

The program scheme is exactly equivalent to the above process, though instead of expanding the regular expression first and then forming the

homomorphism it evaluates the homomorphic image of the regular expression at each step. This means that where it sees ".", "+" or "*" it uses instead the corresponding operation of the particular homomorphism being used, and where it sees a terminal it uses the result of "atom" applied to that micro-instruction. So instead of handling regular expressions the program handles values of the image space of the homomorphism. This is usually much more condensed. It is possible to evaluate the properties of the parts in any order in this fashion, precisely because it is a homomorphism that we are using and so the image of the homomorphism obeys the regular algebra identities.

We turn to considering the order of these manipulations. By examining the microcode the program scheme can distinguish those instructions which can only be entered from one predecessor (usually by a next instruction) from those which may be entered from more than one place. We will call the latter labelled instructions since this is the usual case.

For each of the labelled instructions prepare to store a regular expression, or rather its homomorphic image. Let us for the moment suppose that there are no loops in the microcode and that there is a particular place where the program ends. Then the microcode can be scanned recursively from the distinguished starting instruction using the following algorithm.

```
PROC trace1 = (INSTRUCTION i)IMAGE:
  BEGIN
    PROC scan1 = (INSTRUCTION i)IMAGE:
    BEGIN
      IF is_end(i)
      THEN unit
      ELIF is_next(i)
      THEN dot(atom(i), trace1(i+1))
      ELSE dot(atom(i), plus(trace1(i+1),trace1(dest(i))))
      FI
    END;

    IF is-labelled(i)
    THEN IF empty-memory(i)
        THEN memory(i) := scan1(i)
        FI;
        memory(i)
    ELSE scan1(i)
    FI
  END
```

The algorithm goes down to the end and comes back, applying dot and plus as it returns. But if the instruction is labelled, the value returned to that point is memorised and re-used for subsequent jumps to that label. Accordingly the algorithm traces a spanning tree of the graph of the microcode in a conventional manner. Suppose that the microcode consists of n instructions and that the number of edges in the graph of the microcode is e. The algorithm above works in time proportional to e. If the maximum number of edges leaving an instruction is bounded, and in the kind of microcode that has been described this number in less than or equal to two, then the number of edges is O(n) and so is the time for the algorithm.

We shall now go over to using the modified finite state machine notation described above. What the algorithm will deliver therefore is a rule, though the regular expression parts of those rules will be rendered down into their homomorphic images. The dot operation is therefore re-interpreted as taking one image parameter (the first) and one rule parameter and producing a rule as a result. This rule will be

obtained by prefixing the image parameter to each of the terms in the rule, which we remember has only one occurrence of any non-termainal. The plus operation is likewise re-interpreted to take two rule parameters and deliver a rule. In the result multiple uses of a non-terminal are eliminated by applying the distributive rule. The value, unit, is replaced by the rule with one term consisting of the unit of the homomorphic image followed by a special non-terminal introduced to designate the end. The algorithm above, with IMAGE turned into RULE and the operations changed as described is still an appropriate one for the restricted microcode.

We now allow for the possibility of arbitrary loops, arising from any use of labels and jumps. With each labelled instruction the program also store a flag to tell us whether the instruction is currently being traced, that is, trace2 has been applied to it but this trace has not yet finished. If, while tracing a labelled instruction, the program comes to a jump to that same instruction, it delivers a rule consisting of that instructions non-terminal instead of trying to continue. All such uses of non-terminals point up the spanning tree. After the completion of a scan for each labelled instruction, we look at the resulting rule to see if it contains the non-terminal for the instruction. If it does we modify the result as described above, using the star operation. Since all orders of introducing star like this produce equivalent regular expressions we shall end up with a single term rule with an end non-terminal and a correct homomorphic image.

```
PROC trace2 = (INSTRUCTION i)RULE:
BEGIN
  PROC scan2 = (INSTRUCTION i)RULE:
    BEGIN
      IF is-end(i)
      THEN unit
      ELIF is-next(i)
      THEN dot(atom(i), trace2(i+1))
      ELSE dot(atom(i), plus(trace2(i+1),trace2(dest(i))))
      FI
    END;
```

```
PROC substitute = (RULE r, INSTRUCTION i)RULE:
BEGIN
  RULE result := empty_rule;
  BOOL changed := FALSE;
  FOR k TO UPB r
  DO IF label(name(r[k])) AND
        NOT empty_memory(name(r[k]))
     THEN changed := TRUE;
          result := plus(result,
                         dot(preface_to(name(r[k]),r),
                             substitute(memory(name(r[k]), i))))
     ELSE result := plus(result, r[k])
     FI
  OD;
  IF changed
  THEN memory(i) := result
  FI;
  result
END;
```

```
IF is-labelled(i)
THEN IF empty-memory(i)
     THEN IF active(i)
          THEN non_terminal(i)
          ELSE active(i) := TRUE;
               RULE r = scan2(i);
               active(i) := FALSE;
               RULE res = IF contains(r, non_terminal(i))
                          THEN dot(star(preface_to(i, r)),
                                   rest_of(i, r))
                          ELSE r
                          FI;
               memory(i) := res;
               res
           FI
     ELSE substitute(memory(i), i)
     FI
  ELSE scan2(i)
  FI
END
```

The procedure, substitute, replaces any occurences of the uwpard pointing non-terminals by their definitions if they have become available. The algorithm visits each node just once but at that node it can take a time proportional to the number of jumps into loops, which we call the loop in-degree, k. The recursive call in substitute does not affect this fact. So the total time taken by the algorithm is O(ek), where e is the number of edges in the graph of the microcode. Since the number of edges leaving an instruction is less than or equal to two, this time is O(nk), and if the maximum loop in-degree is also bounded then the algorithm works in O(n). This is the case, for example, for microcode in which all the loops have only one exit and one entry. Real microcode is unlikely to be so well structured.

We move now to consideration of the Decode instruction, which is special to microcode. It is not in fact true that microcode has a starting place and a finishing place. In fact microcode normally loops for ever round a number of macro-instruction interpretations. So the

form of the regular expression for microcode is

preamble.(ins0 + ins1 + ins2 ...)*

Typically a particular kind of control instruction, Decode, starts the interpretation of the next macro-instruction. We therefore generate the regular expression for the whole microcode in the following way. We introduce another special non-terminal which we shall call decode and shall supply for the decode instruction. The end non-terminal we introduced before is therefore not used. Starting at the distinguished instruction we proceed until we have a rule ending just with decode - this is the preamble. Then starting at each of the starting points for macroinstruction decoding we also trace and produce rules ending with decode. These are ins0, ins1 etc. which we add together, star and then dot after the preamble. All this is just a special case of the kind of tracing we did before.

The framework is now set up for discussing how to deal with sub-routines and exceptions. When the algorithm reaches a call instruction, we know that the labelled instruction to which is to pass starts a sub-routine. Accordingly it traces slightly differently. When tracing a sub-routine, if a Decode instruction is reached we have found an error of structure, and this is reported and no value is calculated. If a Return instruction is reached a new special non-terminal is produced. If a Jump-pop instruction is reached a special non-terminal is produced which contains the address to be jumped to. At the end of tracing a correct sub-routine we therefore have a rule which contains some Jump-pop non-terminals and a Return non-terminal. This is what we store with the labelled instruction at the start of the sub-routine. The completion of the processing of a Call is to treat the instruction after the Call as if it were the definition of the Return non-terminal, and to treat each of the Jump-pop non-terminals as if they were Jump non-terminals. This means that the same rule stored at the labelled instruction will suffice for calls performed at different levels in the call stack. At all times the level in the call stack is known and errors involving it are mortal errors of structure.

```
PROC trace2 = (INSTRUCTION i, INT stack_level)RULE:
BEGIN
  PROC scan2 = (INSTRUCTION i)RULE:
    BEGIN
      IF is-next(i)
      THEN dot(atom(i), trace2(i+1, stack_level))
      ELIF is_decode(i)
      THEN IF stack_level = 1
           THEN (atom(i), decode_nt)
           ELSE wrong_structure
           FI
      ELIF is_jump(i)
      THEN dot(atom(i), plus(trace2(i+1, stack_level),
                             trace2(dest(i), stack_level)))
      ELIF is-call(i)
      THEN RULE t = trace(dest(i), stack_level+1),
                q = trace(i+1, stack_level);
           RULE s := dot(preface_to(t, return), q);

         FOR j TO UPB t
         DO IF NOT return_nt(non_terminal(t[j]))
            THEN s := plus(s, dot(terminal(t[j],
                                  trace(jp_dest(non_terminal),
                                        stack_level))))
            FI
         OD;
         s
      ELIF is_return(i)
      THEN IF stack_level > 1
           THEN (atom(i), return)
           ELSE wrong_structure
           FI
      ELIF is_jump_pop(i) AND NOT stack_level = 1
      THEN (atom(i), jump_pop_nt(dest(i)))
      ELSE wrong_structure
      FI
    END;
```

```
PROC substitute = (RULE r, INSTRUCTION i)RULE:
BEGIN
  RULE result := empty_rule;
  BOOL changed := FALSE;
  FOR k TO UPB r
  DO IF label(name(r[k])) AND
        NOT empty_memory(name(r[k]))
     THEN changed := TRUE;
          result := plus(result,
                         dot(preface_to(name(r[k]),r),
                             substitute(memory(name(r[k]), i))))
     ELSE result := plus(result, r[k])
     FI
  OD;
  IF changed
  THEN memory(i) := result
  FI;
  result
END;
```

```
IF is-labelled(i)
THEN IF empty-memory(i)
    THEN IF active(i)
         THEN non-terminal(i)
         ELSE active(i) := TRUE;
              RULE r = scan2(i);
              active(i) := FALSE;
              RULE res = IF contains(r, non_terminal(i))
                         THEN dot(star(preface-to(i, r)),
                                  rest_of(i, r))
                         ELSE r
                         FI;
             memory(i) := res;
             res
          FI
    ELSE substitute(memory(i), i)
    FI
  ELSE scan2(i)
  FI
END
```

The conclusions of the previous argument concerning the time taken by the algorithm still apply, except that there is an additional factor for the maximum number of simultaneous exceptions.

Practical microcode does not obey the one-entry, one-exit rule, and according to the discussion is O(nk). However, in practice k, though not unity, is small.

4 Conclusions

We have described a program scheme for calculating properties of microcode. This program runs quickly and has proved its use in removing errors from about 5000 instructions of ICL Perq microcode implementing the Flex architecture. Many of the tests which were run found errors, some of which were subtle, involving interrupts or

unlikely combinations of circumstance which would have been difficult to remove by the usual methods of trial and error.

References

1 Aho A. V. "Principles of compiler design" Addison-Wesley 1977

2 Allen F.E. and Cocke J. "A program data flow analysis procedure" Comm. ACM Vol 19 No 3 pp 137-147 (1976)

3 Backhouse R. C. and Carré B. A. "Regular algebra applied to path-finding problems" J. Inst. Math. Applic. Vol 15, (1975) pp 161-186

4 Bramson B.D. and Goodenough S.J. "Data use analysis for computer programs", unpublished R.S.R.E. report 1982

5 Bramson B.D. "Information flow analysis for computer programs" unpublished R.S.R.E. report 1982

6 Currie I.F., Edwards P.W. and Foster J.M. "Flex firmware" RSRE Report No. 81009 (1981)

7 Fosdick L.D. and Osterweil L.J. "Data flow analysis in software reliability" Computing Surveys, Vol 8, No 3, pp 305-330, (1976)

8 Foster J.M. "Checking microcode algebraically" R.S.R.E. Memo No. 3748, (1984)

9 Kam J. B. and Ullman J. D. "Monotone data flow analysis frameworks" Acta Inf. Vol 7 (1977) pp 305-317

10 Salomaa A. "Two complete axiom systems for the algebra of regular events" Journal ACM Vol 13, No 1, Jan 1966 pp158-169

11 Tarjan R. E. "A unified approach to path programs"
J. ACM, Vol 28, No 3, July 1981, pp 577-593

12 Wegbreit B. "Property extraction in well-founded property sets"
IEEE Trans. Software Eng. Vol 1 (1975) pp 270-285

13 Carré B.A. "Graphs and networks" OUP 1979

END

FILMED

3-86

DTIC